THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS

Alfred V. Aho
Bell Laboratories

John E. Hopcroft
Cornell University

Jeffrey D. Ullman
Princeton University

Addison-Wesley Publishing Company

Reading, Massachusetts · Menlo Park, California
London · Amsterdam · Don Mills, Ontario · Sydney

The *sorting problem* can be formulated as follows. We are given a sequence of n elements $a_1, a_2, \ldots, a_n$ drawn from a set having a linear order, which we shall usually denote $\leq$. We are to find a permutation π of these n elements that will map the given sequence into a nondecreasing sequence $a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)}$ such that $a_{\pi(i)} \leq a_{\pi(i+1)}$ for $1 \leq i < n$. Usually we shall produce the sorted sequence itself rather than the sorting permutation π.

Sorting methods are classified as being *internal* (where the data resides in random access memory) or *external* (where the data is predominantly outside the random access memory). External sorting is an integral part of such applications as account processing, which usually involve far more elements than can be stored in random access memory at one time. Thus external sorting methods for data which are on secondary storage devices (such as a disk memory or a magnetic tape) have great commercial importance.

Internal sorting is important in algorithm design as well as commercial applications. In those cases where sorting arises as part of another algorithm, the number of items to be sorted is usually small enough to fit in random access memory. However, we assume that the number of items to be sorted is moderately large. If one is going to sort only a handful of items, a simple strategy such as the $O(n^2)$ "bubble sort" (see Exercise 3.5) is far more expedient.

There are numerous sorting algorithms. We make no attempt to survey all the important ones; rather we limit ourselves to methods which we have found to be of use in algorithm design. We first consider the case in which the elements to be sorted are integers or (almost equivalently) strings over a finite alphabet. Here, we see that sorting can be performed in linear time. Then we consider the problem of sorting without making use of the special properties of integers or strings, in which case we are forced to make program branches depend only on comparisons between the elements to be sorted. Under these conditions we shall see that $O(n \log n)$ comparisons are necessary, as well as sufficient, to sort a sequence of n elements.

3.2 RADIX SORTING

To begin our study of integer sorting, let $a_1, a_2, \ldots, a_n$ be a sequence of integers in the range 0 to $m - 1$. If m is not too large, the sequence can easily be sorted in the following manner.

1. Initialize m empty queues, one for each integer in the range 0 to $m - 1$. Each queue is called a *bucket*.
2. Scan the sequence $a_1, a_2, \ldots, a_n$ from left to right, placing element a_i in the a_i-th queue.
3. Concatenate the queues (the contents of queue $i + 1$ are appended to the end of queue i) to obtain the sorted sequence.

Since an element can be inserted into the ith queue in constant time, the n elements can be inserted into the queues in time $O(n)$. Concatenating the m queues requires $O(m)$ time. If m is $O(n)$, then the algorithm sorts n integers in time $O(n)$. We call this algorithm a *bucket sort*.

The bucket sort can be extended to sort a sequence of tuples (i.e., lists) of integers into lexicographic order. Let $\leq$ be a linear order on set S. The relation $\leq$ when extended to tuples whose components are from S is a *lexicographic order* if $(s_1, s_2, \ldots, s_p) \leq (t_1, t_2, \ldots, t_q)$ exactly when either:

1. there exists an integer j such that $s_j < t_j$ and for all $i < j$, $s_i = t_i$, or
2. $p \leq q$ and $s_i = t_i$ for $1 \leq i \leq p$.

For example, if one treats strings of letters (under the natural alphabetic ordering) as tuples, then the words in a dictionary are in lexicographic order.

We first generalize the bucket sort to sequences consisting of k-tuples whose components are integers in the range 0 to $m - 1$. The sorting is done by making k passes over the sequence using the bucket sort on each pass. On the first pass the k-tuples are sorted according to their kth components. On the second pass the resulting sequence is sorted according to the $(k - 1)$st components. On the third pass the sequence resulting from the second pass is sorted according to the $(k - 2)$nd components, and so on. On the kth (and final) pass the sequence resulting from the $(k - 1)$st pass is sorted according to the first components.† The sequence is now in lexicographic order. A precise description of the algorithm is given below.

Algorithm 3.1. Lexicographic sort.

Input. A sequence $A_1, A_2, \ldots, A_n$ where each A_i is a k-tuple

$$(a_{i1}, a_{i2}, \ldots, a_{ik})$$

with a_{ij} an integer in the range 0 to $m - 1$. (A convenient data structure for this sequence of k-tuples is an $n \times k$ array.)

Output. A sequence $B_1, B_2, \ldots, B_n$ which is a permutation of $A_1, A_2, \ldots, A_n$ such that $B_i \leq B_{i+1}$ for $1 \leq i < n$.

Method. In transferring a k-tuple A_i to some bucket, only a pointer to A_i is actually moved. Thus A_i can be added to a bucket in fixed time rather than time bounded by k. We use a queue called QUEUE to hold the "current" sequence of elements. An array Q of m buckets is also used, where bucket $Q[i]$ is intended to hold those k-tuples that have the integer i in the component currently under consideration. The algorithm is shown in Fig. 3.1. □

† In many practical situations it is sufficient to bucket sort the strings only on the basis of their first components. If the number of elements that get placed into each bucket is small, then we can sort the strings in each bucket with some straightforward sorting algorithm such as bubble sort.

```
begin
    place A_1, A_2, ..., A_n in QUEUE;
    for j ← k step −1 until 1 do
        begin
            for l ← 0 until m − 1 do make Q[l] empty;
            while QUEUE not empty do
                begin
                    let A_i be the first element in QUEUE;
                    move A_i from QUEUE to bucket Q[a_ij]
                end;
            for l ← 0 until m − 1 do
                concatenate contents of Q[l] to the end of QUEUE
        end
end
```

Fig. 3.1. Lexicographic sort algorithm.

Theorem 3.1. Algorithm 3.1 lexicographically sorts a length n sequence of k-tuples, where each component of a k-tuple is an integer between 0 and $m - 1$, in time $O((m + n)k)$.

Proof. The proof that Algorithm 3.1 works correctly is by induction on the number of executions of the outer loop. The induction hypothesis is that after r executions, the k-tuples in QUEUE will be lexicographically sorted according to their r rightmost components. The result is easily established once it is observed that the $(r + 1)$st execution sorts k-tuples by their $(r + 1)$st component from the right, and that if two k-tuples are placed in the same bucket, the first k-tuple precedes the second in the lexicographic order determined by the r rightmost components.

One pass of the outer loop of Algorithm 3.1 requires $O(m + n)$ time. The loop is repeated k times to give a time complexity of $O((m + n)k)$. □

Algorithm 3.1 has a variety of applications. It has been used in punched card sorting machines for a long time. It can also be used to sort $O(n)$ integers in the range 0 to $n^k - 1$ in time $O(kn)$, since such an integer can be thought of as a k-tuple of digits in the range 0 to $n - 1$ (i.e., the representation of the integer in base n notation).

Our final generalization of the bucket sort will be to tuples of varying sizes, which we shall call strings. If the longest string is of length k, we could pad out every string with a special symbol to make all strings be of length k and then use Algorithm 3.1. However, if there are only a few long strings, then this approach is unnecessarily inefficient for two reasons. First, on each pass, every string is examined, and second, every bucket $Q[i]$ is examined even if almost all of these buckets are empty. We shall describe an algorithm that sorts a sequence of n strings of varying length, whose components are in

the range 0 to $m-1$, in time $O(m+l_{\text{total}})$, where l_i is the length of the ith string, and $l_{\text{total}} = \sum_{i=1}^{n} l_i$. The algorithm is useful in the situation where m and l_{total} are both $O(n)$.

The essence of the algorithm is to first arrange the strings in order of decreasing length. Let $l_{\max}$ be the length of the longest string. Then, as in Algorithm 3.1, $l_{\max}$ passes of the bucket sort are used. However, the first pass sorts (by rightmost component) only those strings of length $l_{\max}$. The second pass sorts [according to the $(l_{\max}-1)$st component] those strings of length at least $l_{\max}-1$, and so on.

For example, suppose *bab*, *abc*, and *a* are three strings to be sorted. (We have assumed that the components of tuples are integers, but for notational convenience we shall often use letters instead. This should cause no difficulty because we can always substitute 0, 1, and 2 for *a*, *b*, and *c*, if we like.) Here $l_{\max} = 3$, so on the first pass we would sort only the first two strings on the basis of their third components. In sorting these two strings we would put *bab* into the *b*-bucket and *abc* into the *c*-bucket. The *a*-bucket would remain empty. In the second pass we would sort these same two strings on the second component. Now the *a*-bucket and *b*-bucket would be occupied, but the *c*-bucket would remain empty. On the third and final pass we would sort all three strings on the basis of their first component. This time the *a*- and *b*-buckets would be occupied and the *c*-bucket would be empty.

We can see that in general on a given pass many buckets can be empty. Thus a preprocessing step that determines which buckets will be nonempty on a given pass is beneficial. The list of nonempty buckets for each pass is determined in increasing order of bucket number. This allows us to concatenate the nonempty buckets in time proportional to the number of nonempty buckets.

Algorithm 3.2. Lexicographic sort of strings of varying length.

Input. A sequence of strings (tuples), $A_1, A_2, \ldots, A_n$, whose components are integers in the range 0 to $m-1$. Let l_i be the length of $A_i = (a_{i1}, a_{i2}, \ldots, a_{il_i})$, and let $l_{\max}$ be the largest of the l_i's.

Output. A permutation $B_1, B_2, \ldots, B_n$ of the A_i's such that

$$B_1 \leq B_2 \leq \cdots \leq B_n.$$

Method

1. We begin by making lists, one for each l, $1 \leq l \leq l_{\max}$, of those symbols that appear in the lth component of one or more of the strings. We do so by first creating, for each component a_{il}, $1 \leq i \leq n$, $1 \leq l \leq l_i$ of each string A_i, a pair (l, a_{il}). Such a pair indicates that the lth component of some string contains the integer a_{il}. These pairs are then sorted lex-

```
        begin
 1.         make QUEUE empty;
 2.         for j ← 0 until m − 1 do make Q[j] empty;
 3.         for l ← l_max step −1 until 1 do
                begin
 4.                 concatenate LENGTH[l] to the beginning of
                      QUEUE;†
 5.                 while QUEUE not empty do
                        begin
 6.                         let A_i be the first string on QUEUE;
 7.                         move A_i from QUEUE to bucket Q[a_il]
                        end;
 8.                 for each j on NONEMPTY[l] do
                        begin
 9.                         concatenate Q[j] to the end of QUEUE;
10.                         make Q[j] empty
                        end
                end
        end
```

† Technically, we should only concatenate to the end of a queue, but concatenation to the beginning should present no conceptual difficulty. The most efficient approach here would be to select A_i's from LENGTH[l] first at line 6, then select them from QUEUE, without ever concatenating LENGTH[l] and QUEUE at all.

Fig. 3.2. Lexicographic sort of strings of varying length.

icographically by an obvious generalization of Algorithm 3.1.‡ Then, by scanning the sorted list from left to right it is easy to create l_{max} sorted lists NONEMPTY[l], for $1 \le l \le l_{max}$, such that NONEMPTY[l] contains exactly those symbols that appear in the lth component of some string. That is, NONEMPTY[l] contains, in sorted order, all those integers j such that $a_{il} = j$ for some i.

2. We determine the length of each string. We then make lists LENGTH[l], for $1 \le l \le l_{max}$, where LENGTH[l] consists of all strings of length l. (Although we speak of moving a string, we are only moving a pointer to a string. Thus each string can be added to LENGTH[l] in a fixed number of steps.)
3. We now sort the strings by components as in Algorithm 3.1, beginning with the components in position l_{max}. However, after the ith pass

‡ In Algorithm 3.1, the assumption was made that components were chosen from the same alphabet. Here, the second component ranges from 0 to $m - 1$, while the first ranges from 1 to l_{max}.

QUEUE contains only those strings of length $l_{max} - i + 1$ or greater, and these strings will already be sorted according to components $l_{max} - i + 1$ through l_{max}. The NONEMPTY lists computed in step 1 are used to determine which buckets are occupied at each pass of the bucket sort. This information is used to help speed up the concatenation of the buckets. This part of the algorithm is given in Pidgin ALGOL in Fig. 3.2 on p. 81. □

Example 3.1. Let us sort the strings *a*, *bab*, and *abc* using Algorithm 3.2. One possible representation for these strings is the data structure shown in Fig. 3.3. STRING is an array such that STRING[i] is a pointer to the representation of the ith string whose length and components are stored in the array DATA. The cell in DATA pointed to by STRING[i] gives the number j of symbols in the ith string. The next j cells of DATA contain these symbols.

The lists of strings used by Algorithm 3.2 are really lists of pointers such as those in the array STRING. For notational convenience, in the remainder of this example we shall write the actual strings, rather than the pointers to the strings, on lists. Bear in mind, however, that it is the pointers rather than the strings themselves that are being stored in the queues.

In part 1 of Algorithm 3.2 we create the pair $(1, a)$ from the first string, the pairs $(1, b)$, $(2, a)$, $(3, b)$ from the second, and the pairs $(1, a)$, $(2, b)$, $(3, c)$ from the third. The sorted list of these pairs is:

$$(1, a)\ (1, a)\ (1, b)\ (2, a)\ (2, b)\ (3, b)\ (3, c).$$

By scanning this sorted list from left to right we deduce that

$$\text{NONEMPTY}[1] = a, b$$
$$\text{NONEMPTY}[2] = a, b$$
$$\text{NONEMPTY}[3] = b, c$$

In part 2 of Algorithm 3.2 we compute $l_1 = 1$, $l_2 = 3$, and $l_3 = 3$. Thus LENGTH[1] $= a$, LENGTH[2] is empty, and LENGTH[3] $=$ *bab*, *abc*. We therefore begin part 3 by setting QUEUE $=$ *bab*, *abc*, and sorting these strings by their third component. The fact that NONEMPTY[3] $= b, c$ assures us that when we form the sorted list in lines 8–10 of Fig. 3.2, $Q[a]$ need not be concatenated to the end of QUEUE. We thus have QUEUE $=$ *bab*, *abc* after the first pass of the loop of lines 3–10 in Fig. 3.2.

In the second pass, QUEUE does not change, since LENGTH[2] is empty and the sort by second component does not change the order. In the third pass, we set QUEUE to *a*, *bab*, *abc* at line 4. The sort by first components gives QUEUE $=$ *a*, *abc*, *bab*, which is the correct order. Note that in the third pass, $Q[c]$ remains empty, and since c is not on NONEMPTY[1], we do not concatenate $Q[c]$ to the end of QUEUE. □

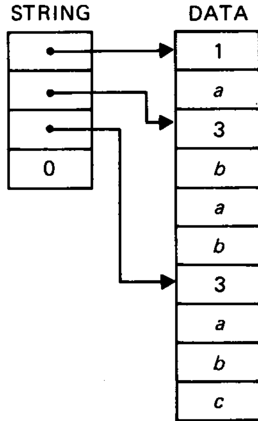

Fig. 3.3 Data structure for strings.

Theorem 3.2. Algorithm 3.2 sorts its input in time $O(l_{\text{total}} + m)$, where

$$l_{\text{total}} = \sum_{i=1}^{n} l_i.$$

Proof. An easy induction on the number of passes through the outer loop in Fig. 3.2 proves that after i passes, QUEUE contains those strings of length $l_{\max} - i + 1$ or greater, and that they are sorted according to components $l_{\max} - i + 1$ through $l_{\max}$. Thus the algorithm lexicographically sorts its input.

For the timing result, part 1 of the algorithm uses $O(l_{\text{total}})$ time to create the pairs and $O(m + l_{\text{total}})$ time to sort them. Similarly, part 2 requires no more than $O(l_{\text{total}})$ time.

We must now direct our attention to part 3 and the program of Fig. 3.2. Let n_i be the number of strings having an ith component. Let m_i be the number of different symbols appearing in the ith components of the strings (i.e. m_i is the length of NONEMPTY[i]).

Consider a fixed value of l in line 3 of Fig. 3.2. The loop of lines 5–7 requires $O(n_l)$ time and the loop of lines 8–10 requires $O(m_l)$ time. Step 4 requires constant time, so one pass through the loop of lines 3–10 requires $O(m_l + n_l)$ time. Thus the entire loop takes

$$O\left(\sum_{l=1}^{l_{\max}} (m_l + n_l)\right)$$

time. Since

$$\sum_{l=1}^{l_{\max}} m_l \le l_{\text{total}} \qquad \text{and} \qquad \sum_{l=1}^{l_{\max}} n_l = l_{\text{total}},$$

we see that lines 3–10 require time $O(l_{\text{total}})$. Then, as line 1 requires constant time and line 2 requires $O(m)$ time, we have the desired result. □

We offer one example where string sorting arises in the design of an algorithm.

Example 3.2. Two trees are said to be *isomorphic* if we can map one tree into the other by permuting the order of the sons of vertices. Consider the problem of determining whether two trees T_1 and T_2 are isomorphic. The following algorithm works in time linearly proportional to the number of vertices. The algorithm assigns integers to the vertices of the two trees, starting with vertices at level 0 and working up towards the roots, in such a way that the trees are isomorphic if and only if their roots are assigned the same integer. The algorithm proceeds as follows.

1. Assign to all leaves of T_1 and T_2 the integer 0.
2. Inductively, assume that all vertices of T_1 and T_2 at level $i - 1$ have been assigned integers. Assume L_1 is a list of the vertices of T_1 at level $i - 1$ sorted by nondecreasing value of the assigned integers. Assume L_2 is the corresponding list for T_2.†
3. Assign to the nonleaves of T_1 at level i a tuple of integers by scanning the list L_1 from left to right and performing the following actions: For each vertex v on list L_1 take the integer assigned to v to be the next component of the tuple associated with the father of v. On completion of this step, each nonleaf w of T_1 at level i will have a tuple $(i_1, i_2, \ldots, i_k)$ associated with it, where $i_1, i_2, \ldots, i_k$ are the integers, in nondecreasing order, associated with the sons of w. Let S_1 be the sequence of tuples created for the vertices of T_1 on level i.
4. Repeat step 3 for T_2 and let S_2 be the sequence of tuples created for the vertices of T_2 on level i.
5. Sort S_1 and S_2 using Algorithm 3.2. Let S_1' and S_2', respectively, be the sorted sequences of tuples.
6. If S_1' and S_2' are not identical, then halt; the trees are not isomorphic. Otherwise, assign the integer 1 to those vertices of T_1 on level i represented by the first distinct tuple on S_1', assign the integer 2 to the vertices represented by the second distinct tuple, and so on. As these integers are assigned to the vertices of T_1 on level i, make a list L_1 of the vertices so assigned. Append to the front of L_1 all leaves of T_1 on level i. Let L_2 be the corresponding list of vertices of T_2. These two lists can now be used for the assignment of tuples to vertices at level $i + 1$ by returning to step 3.

† You should convince yourself that level numbers can be assigned in $O(n)$ steps by a preorder traversal of the tree.